Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava and Julien Tesson

Laboratory of Algorithmic Complexity and Logic
Université Paris-Est

HLPP July 2015
Table of Contents

1 Introduction
2 Multi-ML
3 Results
4 Conclusion
Table of Contents

1 Introduction
   BSP
   BSML
   MULTI-BSP

2 Multi-ML

3 Results

4 Conclusion
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- Pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution
- Confluent
- Deadlock-free
- Predictable performances
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- \( p \) pairs CPU/memory
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- $p$ pairs CPU/memory
- Communication network
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- \( p \) pairs CPU/memory
- Communication network
- Synchronization unit
Bulk Synchronous Parallelism

The BSP computer

Defined by:

- $p$ pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution

local computations
communication barrier
next super-step

V. Allombert et al.

HLPP 2015

2 / 17
Bulk Synchronous Parallelism

The BSP computer
Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronization unit

Properties:
- Super-steps execution
- Confluent
**Bulk Synchronous Parallelism**

**The BSP computer**

Defined by:
- $p$ pairs CPU/memory
- Communication network
- Synchronization unit

**Properties:**
- Super-steps execution
- Confluent
- Deadlock-free
Bulk Synchronous Parallelism

The BSP computer

Defined by:
- $p$ pairs CPU/memory
- Communication network
- Synchronization unit

Properties:
- Super-steps execution
- Confluent
- Deadlock-free
- Predictable performances

V. Allombert et al.
HLPP 2015
What is BSML?

• Explicit BSP programming with a functional approach
• Based upon ML implemented over OCaml
• Formal semantics → computer-assisted proofs (Coq)

Main idea: Parallel data structure ⇒ vectors: V. Allombert et al. HLPP 2015
What is BSML?

- Explicit BSP programming with a functional approach
Bulk Synchronous ML

What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
**Bulk Synchronous ML**

### What is BSML?

- Explicit **BSP** programming with a functional approach
- Based upon **ML** an implemented over **OCAML**
- Formal semantics → computer-assisted proofs (**COQ**)
Bulk Synchronous ML

What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal semantics → computer-assisted proofs (COQ)

Main idea

Parallel data structure ⇒ vectors:
BSML primitives

Asynchronous primitives

• $\langle e, \ldots, e \rangle$

• $v$ : $v$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$

• $\text{pid} : A$ predefined vector: $i$ on processor $i$

Synchronous primitives

• $\text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i)$

• $\text{put} : \langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle (\text{fun } i \rightarrow f_i(0)), \ldots, (\text{fun } i \rightarrow f_i(p-1)) \rangle$
BSML primitives

Asynchronous primitives

- $\ll e \gg : \langle e, \ldots, e \rangle$
BSML primitives

Asynchronous primitives

- $\langle e \rangle$ : $\langle e, \ldots, e \rangle$
- $v_i$ : $v_i$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
Asynchronous primitives

- \( \ll e \gg : \langle e, \ldots, e \rangle \)
- \( v \) : \( v_i \) on processor \( i \), assumes \( v \equiv \langle v_0, \ldots, v_{p-1} \rangle \)
- \( \text{pid} \) : A predefined vector: \( i \) on processor \( i \)
**BSML primitives**

### Asynchronous primitives

- $\ll e \gg$ : $\langle e, \ldots, e \rangle$
- $v_i$ : $v_i$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- $\text{pid}_i$ : A predefined vector: $i$ on processor $i$

### Synchronous primitives

- $\text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i)$
Asynchronous primitives

- $\ll e \gg : \langle e, \ldots, e \rangle$
- $\nu$ : $v_i$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- $\text{pid}$ : A predefined vector: $i$ on processor $i$

Synchronous primitives

- $\text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \to x_i)$
- $\text{put} : \langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle (\text{fun } i \to f_i \ 0), \ldots, (\text{fun } i \to f_i \ (p-1)) \rangle$
For a BSP machine with 3 processors:

```ocaml
# let vec = "HLPP
" ;;
val vec : string par = <"HLPP", "HLPP", "HLPP">

# let vec2 = $vec^$(string_of_int $pid$) ;;
val vec2 : string par = <"HLPP_0", "HLPP_1", "HLPP_2">

# let totex v = List.map (proj v) procs;;
val totex : 'a Bsml.par → 'a list = <fun>

# totex vec2;;
— : string list = ["HLPP0"; "HLPP1"; "HLPP2"]
```
The **MULTI-BSP** model

**What is MULTI-BSP?**
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors

MULTI-BSP

```
Multi-Core
  core0
    th0 th1 th2 th3
  core1
    th0 th1 th2 th3
```
What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The **MULTI-BSP model**

**Execution model**

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges information with the $m_i$ memory
- Synchronizes Level $i$ with Level $i - 1$

\[ \ldots \]

V. Allombert et al.

HLPP 2015
A level $i$ superstep is:

- Level $i - 1$ executes code independently

The MULTI-BSP model
The **MULTI-BSP** model

**Execution model**

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
The MULTI-BSP model

Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
- Synchronises
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Multi-ML</td>
</tr>
<tr>
<td></td>
<td>Overview</td>
</tr>
<tr>
<td></td>
<td>Primitives</td>
</tr>
<tr>
<td></td>
<td>Semantics</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
</tr>
<tr>
<td>3</td>
<td>Results</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>
Basic ideas:

- Multi-ml-like code on every stage of the multi-bsp architecture
- Specific syntax over ml: eases programming

\[
\begin{align*}
\text{let } v = \langle e \rangle \\
\end{align*}
\]
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the tree.
Basic ideas:

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the tree.
Recursion structure

let multi f [args] =
    where node =
        (* BSML code *)
        ...
    ≪ f [args] ≫
    ...

where leaf =
    (* OCAML code *)
    ...

Recursion structure

```
let multi f [args] =
   where node =
      (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
   where leaf =
      (* OCAML code *)
   ...
```
Recursion structure

```
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ...
  where leaf =
    (* OCAML code *)
    ...
```

MULTI-ML
Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ...
  where leaf =
    (* OCAML code *)
    ...
```

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
  ...
  ≪ f [args] ≫
  ...
where leaf =
  (* OCAML code *)
  ...
```

MULTI-ML
Recursion structure

let multi f [args] =
where node =
(* BSML code *)
...
≪ f [args] ≫
...
where leaf =
(* OCAML code *)
...

MULTI-ML
Recursion structure

\[ \text{let multi } f \ [\text{args}] = \]
\[ \text{where node } = \]
\[ (* \text{ BSML code } *) \]
\[ \ldots \]
\[ \ll f \ [\text{args}] \gg \]
\[ \ldots \]
\[ \text{where leaf } = \]
\[ (* \text{ OCAML code } *) \]
\[ \ldots \]
Recursion structure

```ocaml
let multi f [args] =
where node =
  (* BSML code *)
...
≪ f [args] ≫
...
where leaf =
  (* OCAML code *)
...
```

MULTI-ML
Recursion structure

let multi f [args] =  
where node =  
(* BSML code *)  
...  
≪ f [args] ≫  
...  
where leaf =  
(* OCAML code *)  
...
Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
  ...
  ≪ f [args] ≫
  ...
  where leaf =
    (* OCAML code *)
  ...
```

MULTI-ML
Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
  ...
  ≪ f [args] ≫
  ...
  where leaf =
    (* OCAMLR code *)
  ...
```

Result

```
```

MULTI-ML
Primitives

Summary:
Primitives

Summary:

- $\varepsilon e \varepsilon$

Diagram: (Graphical representation of a tree with nodes labeled $e$)
Primitives

Summary:

- \$e\$
- gid

Diagram:

```
  0
 /\  /
0.0 0.1
 / \ /  \
0.0.0 0.0.1 0.1.0 0.1.1
```
Primitives

Summary:

- $e$
- $\text{gid}$
- $\text{at}$
Primitives

Summary:

- $e$
- gid
- at

\[ \text{at } v \]

\[ v_{0.1} \]
Primitives

Summary:

- $e$
- gid
- at
- $\langle\ldots f\ldots\rangle$
Primitives

Summary:

- $\mathfrak{e}$
- gid
- at
- $\ll\ldots f\ldots\gg$

\begin{center}
\begin{tikzpicture}
  \node (N) {$\mathcal{N}$} child { node (f) {f} } child { node (f) {f} };
\end{tikzpicture}
\end{center}
Summary:
- $x\in\mathbb{R}$
- gid
- at
- $\ll\ldots f\ldots\gg$
- $\#\times\#$
Primitives

Summary:

- \( e \)
- \( gid \)
- \( at \)
- \( \langle...f...\rangle \)
- \( \#x\# \)
Summary:

- \$e\$
- gid
- at
- \(\ll...f...\rr\)
- \#\times\#
- mkpar \(f\)

Primitives

\[
\text{mkpar (fun } i \rightarrow v_i)\]

V. Allombert et al.

HLPP 2015
Summary:

- $\mathbb{e}$
- gid
- at
- $\ll \ldots f \ldots \rr$
- $\#x\#$
- mkpar $f$

Primitives

```
finally v1, v2
mkpar (fun i -> vi)
```
Primitives

Summary:
- $\texttt{§e§}$
- $\texttt{gid}$
- $\texttt{at}$
- $\lhd \ldots \ldots \rhd$
- $\texttt{#x#}$
- $\texttt{mkpar} \ f$

```
finally

mkpar (fun i \rightarrow v_i)
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```

```
\texttt{mkpar (fun i \rightarrow v_i)}
```
Primitives

Summary:

- $\varepsilon\varepsilon$
- gid
- at
- $\ll\ldots f \ldots\gg$
- $\#\times\#$
- mkpar f
- finally $v_1 v_2$
Primitives

Summary:

- \$e\$
- gid
- at
- \langle...f...\rangle
- \#x\#
- mkpar f
- finally \(v_1 \ v_2\)
Summary:

- $\emptyset e \emptyset$
- gid
- at
- $\ll \ldots f \ldots \gg$
- $\# x \#$
- mkpar f
- finally $v_1 v_2$
Primitives

Summary:

- \$e\$
- gid
- at
- \(<...f...>
- \#\#\#
- mkpar f
- finally \texttt{v1 v2}
- this
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$) ) in
  finally ~up:s ~keep:s
where leaf =
  let s = sumSeq l in
  finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in
    finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in
    finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```ocaml
let multi tree sum_list l =  
  where node =  
  let v = mkpar (fun i → split i l) in  
  let s = sumSeq (flatten ≪ sum_list $v$ ) in  
  finally ~up:s ~keep:s

where leaf =  
  let s = sumSeq l in  
  finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in
    finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

\[
\text{let multi tree sum_list } l = \\
\text{where node =}
\]
\[
\text{let } v = \text{mkpar (fun } i \rightarrow \text{split } i \text{ } l) \text{ in}
\]
\[
\text{let } s = \text{sumSeq (flatten } \ll\text{ sum_list } v \rr\text{ ) in}
\]
\[
\text{finally } \text{up:} s \text{ } \text{keep:} s
\]

\[
\text{where leaf =}
\]
\[
\text{let } s = \text{sumSeq } l \text{ in}
\]
\[
\text{finally } \text{up:} s \text{ } \text{keep:} s
\]
Keep the intermediate results of the sum:

```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$) ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in
    finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```ocaml
let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let s = sumSeq (flatten (\sum_list v)) in  
    finally ~up:s ~keep:s  
  where leaf =  
    let s = sumSeq l in  
    finally ~up:s ~keep:s
```
Keep the intermediate results of the sum:

```plaintext
let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let s = sumSeq (flatten ≪ sum_list $v$) in  
    finally ~up:s ~keep:s  
  where leaf =  
    let s = sumSeq l in  
    finally ~up:s ~keep:s
```

Keep the intermediate results of the sum:

```ocaml
let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i → split i l) in  
    let s = sumSeq (flatten ≪ sum_list $v$) ) in  
  finally ~up:s ~keep:s  
where leaf =  
  let s = sumSeq l in  
  finally ~up:s ~keep:s
```
Semantics

Formal definition of a core-language

Useful for:
Formal definition of a core-language

Useful for:
- Study of properties
Semantics

Formal definition of a core-language

Useful for:

- Study of properties
- Proof of programs/compiler/typing rules

V. Allombert et al.
HLPP 2015
Semantics

Formal definition of a core-language

Useful for:
- Study of properties
- Proof of programs/compilertyping rules

Currently
- Inductive big-step: confluent
- Co-inductive: mutually exclusive
Implementation

Sequential simulator

- OCAML-like toplevel
- Test and debug
- Tree structure
- Hash tables to represent memories

```ocaml
#let multi f n =
  where node =
    let _=<<(f ($pid$ + #n# + 1) >> in
    finally ~up:() ~keep:(gid” =” ^n)
  where leaf=finally ~up:() ~keep:(gid” =” ^n);

— : val f : int→string tree = <multi-fun>
#(f 0)
o "0→ 0"
  |— o "0.0→ 1"
    |— o "0.0.0→ 2"
    |— o "0.0.1→ 3"
  |— o "0.1→ 2"
    |— o "0.1.0→ 3"
    |— o "0.1.1→ 4"
```

V. Allombert et al.

HLPP 2015
Distributed implementation

Our approach
Distributed implementation

<table>
<thead>
<tr>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modular</td>
</tr>
<tr>
<td>Distributed implementation</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td><strong>Our approach</strong></td>
</tr>
<tr>
<td>• Modular</td>
</tr>
<tr>
<td>• Generic functors</td>
</tr>
</tbody>
</table>
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
## Distributed implementation

### Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories
### Distributed implementation

#### Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

#### Current version

- Based on MPI
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
### Distributed implementation

#### Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

#### Current version

- Based on MPI
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations
Table of Contents

1 Introduction
2 Multi-ML
3 Results
4 Conclusion
Naive Eratosthenes algorithm

- \( \sqrt{n} \)th first prime numbers
- Based on scan
- Unbalanced
Naive Eratosthenes algorithm

- $\sqrt{n}$th first prime numbers
- Based on scan
- Unbalanced

Benchmarks

Mirev 3
Benchmarks

**Naive Eratosthenes algorithm**
- $\sqrt{(n)}$th first prime numbers
- Based on scan
- Unbalanced

**Results**

<table>
<thead>
<tr>
<th></th>
<th>100_000</th>
<th></th>
<th>500_000</th>
<th></th>
<th>1_000_000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
</tr>
<tr>
<td>8</td>
<td>0.7</td>
<td>1.8</td>
<td>22.4</td>
<td>105.0</td>
<td>125.3</td>
</tr>
<tr>
<td>64</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
<td>8.7</td>
<td>4.1</td>
</tr>
<tr>
<td>128</td>
<td>0.5</td>
<td>0.45</td>
<td>2.1</td>
<td>5.2</td>
<td>4.7</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction
2. Multi-ML
3. Results
4. Conclusion
Conclusion

MULTI-ML

Recursive multi-functions
Structured nesting of bsml codes
Big-steps formal semantics (confuent)
Small number of primitives and little syntax extension

Future work
Optimise mpi implementation
Type system for multi-ml
Real life benchmarks
Conclusion

MULTI-ML

- Recursive multi-functions
## Conclusion

### MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Small number of primitives and little syntax extension

Future work

- Optimise mpi implementation
- Type system for multi-ml
- Real life benchmarks
Conclusion

**MULTI-ML**

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Small number of primitives and little syntax extension

**Future work**

- Optimise MPI implementation
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Small number of primitives and little syntax extension

Future work

- Optimise MPI implementation
- Type system for MULTI-ML
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Small number of primitives and little syntax extension

Future work

- Optimise MPI implementation
- Type system for MULTI-ML
- Real life benchmarks
Thank you for your attention!

Any questions?